Universidad Nacional de la Plata

Tesis de Licenciatura en Informática

Automatización del proceso de generación de Casos de Prueba para Casos de Uso en UML.

Carlos Alberto Sokol

Directora de Tesis: Prof. Mg. Roxana Giandini

Clasificación: Investigación Aplicada

Octubre de 2007

Agradecimientos:

Quiero agradecer a todos los que me apoyaron y dieron fuerzas para realizar este proyecto, comenzando por mis padres, principales motivadores de mis logros, a Patricia y amigos que me mostraron especial paciencia, y a mi directora de tesis quien me ayudó enormemente y animó cuando las circunstancias no eran del todo favorables.

Resumen:

Este trabajo consta de una investigación de los aspectos necesarios para la generación automática de Casos de Prueba, y la presentación de un prototipo que permite diversas configuraciones para obtener en un formato legible para humanos, diferentes tipos de Casos de Pruebas según se necesite, para auxiliar la tarea de testing humano y la codificación posterior de las pruebas de unidad por desarrolladores, contribuyéndose a la calidad en los procesos de IS.

Tabla de Contenidos:

1. Introducción

El propósito principal de la *Ingeniería de Software* es construir software de calidad, idea aceptada tanto en el ámbito académico como en el de la industria.
Para lograr esto, se busca calidad en los componentes, y la reutilización de software.
Con la reutilización, se obtienen aspectos positivos como la mejora en tiempos de desarrollo, pero se generan errores propios de esta práctica, que se suman a los errores del desarrollo en sí, y por eso la gran importancia del testing.

La construcción de un sistema de software debe ser acompañada en todo momento por el área de Quality Assurance (QA), para detectar los defectos en el desarrollo lo más tempranamente posible, ya que el costo de solucionar "diferencias" entre lo que se construye y lo que realmente se necesita (bugs o errores), se encarece cuanto más nos alejamos de la etapa de análisis y nos acercamos a la etapa de implementación.

Los *Casos de Prueba* son la columna vertebral de QA, pues se manifiestan en todo el ciclo de vida del proyecto como pruebas de Unidad, Funcionales, de Integración, de Módulo, etc.

Los mismos suelen diseñarse en documentación estática como documentos de Word o Excel.

Los cambios de requerimientos son muy comunes, como también lo son los ajustes en el análisis, se torna difícil mantener los Casos de Pruebas diseñados en forma manual, que es la forma usual en muchas empresas actualmente.

Sin embargo, el testing no se lo aplica extensivamente como se debiera en la industria de hoy en día. Se aplica lo mínimo y se la toma como una actividad cara y de poca ganancia.
Las compañías buscan que el software sea “lo suficientemente bueno” para entregar, aunque contenga muchos bugs.

En estadísticas, se observa que *se pierden millones por no testear lo suficiente*, y esto sin contemplar los costos de los errores cuando significan perdidas de vidas o catástrofes. La falta de métricas de calidad hace que en muchos sitios tan solo se cuenten los bugs. [DesignByContract]

Un tester efectivo es el que tiene un buen conjunto de técnicas de testing, entiende como el producto se comporta en el entorno operativo, tiene olfato para probar donde los más oscuros bugs pueden aparecer, y utiliza el conjunto de técnicas para detectarlos.

Dijkstra dijo en 1972 que programas de testing se pueden usar para detectar errores, pero nunca su ausencia. [Dijkstra]

Por esto además podemos concentrarnos en que la cobertura sea óptima, para detectar la mayor cantidad posible.

No se puede testear al 100 por ciento, por ende es muy importante tomar una muestra significativa de las variaciones de prueba para ejecutar esas variaciones solamente, y si se lo hace efectivamente, se logra cubrir la mayor cantidad de bugs. [op-profile]

1.1. Organización de la tesis

En el capítulo 1 se introducen los conceptos fundamentales de QA, testing y su importancia para la IS.

En el capítulo 2 se explican los Casos de Uso de UML para modelar los procesos de negocio, que dan el punto de entrada para analizar los requisitos del sistema, y los problemas que se necesitan solucionar.
Luego se explican los Casos de Pruebas, documentación del testing, que son los ladrillos fundamentales de QA y las distintas configuraciones necesarias para cada tipo de Caso de Prueba.

En el capítulo 3 se enumeran las ventajas de la generación automática de los Casos de Prueba.

En el capítulo 4 se detalla el objetivo de la tesis, que se puede resumir en la generación de Casos de Prueba en forma automática y configurable para contribuir a la calidad en la Ingeniería de Software.

En el capítulo 5 se describe la propuesta: Presentación de un aplicativo prototipo para crear variaciones de Casos de Prueba que sirva para mostrar prácticamente las conclusiones analizadas.

En el capítulo 6 se detalla la metodología y cada etapa para llegar a la implementación, y finalmente los resultados obtenidos.

En el capítulo 7 se dan las conclusiones y los puntos a extender para trabajos futuros.

2. Conceptos

UML está siendo ampliamente usado hoy en día en la Ingeniería de Software.

UML (Lenguaje Unificado de Modelado) describe un conjunto de notaciones y diagramas estándar para modelar sistemas orientados a objetos, y describe la semántica esencial de lo que estos diagramas y símbolos significan. UML reemplaza a las muchas notaciones y métodos usados para el diseño orientado a objetos, pues es una evolución de muchos de ellos.

Con UML, los modeladores sólo tienen que aprender una única notación.

UML se puede usar para modelar distintos tipos de sistemas: sistemas de software, sistemas de hardware, y organizaciones del mundo real.

UML es una consolidación de muchas de las notaciones y conceptos más usados orientados a objetos. Empezó como una consolidación del trabajo de Grado Booch, James Rumbaugh, e Ivar Jacobson, creadores de tres de las metodologías orientadas a objetos más populares.

En 1996, el Object Management Group (OMG), un pilar estándar para la comunidad del diseño orientado a objetos, publicó una petición con propósito de un metamodelo orientado a objetos de semántica y notación estándares. UML, en su versión 1.0, fue propuesto como una respuesta a esta petición en enero de 1997. Hubo otras cinco propuestas rivales. Durante el transcurso de 1997, los seis promotores de las propuestas, unieron su trabajo y presentaron al OMG un documento revisado de UML, llamado UML versión 1.1. Este documento fue aprobado por el OMG en Noviembre de 1997. El OMG llama a este documento OMG UML versión 1.1.

UML describe una notación estándar y semántica esencial para el modelado de un sistema orientado a objetos. Previamente, un diseño orientado a objetos podría haber sido modelado con cualquiera de la docena de metodologías populares,

causando a los revisores tener que aprender las semánticas y notaciones de la metodología empleada antes que intentar entender el diseño en sí. Con UML, diseñadores diferentes modelando sistemas diferentes pueden entender cada uno los diseños de los otros. [UML]

UML es una notación, pero no es un proceso o método estándar para desarrollar un sistema. Hay varias metodologías existentes, siendo [RUP] una de las más populares.

Muchas organizaciones han desarrollado sus propias metodologías internas, usando diferentes diagramas y técnicas con orígenes varios. Ejemplos son el método Catalyst por Computer Sciences Corporation (CSC) o el Worldwide Solution Design and Delivery Method (WSDDM) por IBM. Estas metodologías difieren, pero generalmente combinan análisis de flujo de trabajo, captura de los requisitos, y modelado de negocio con modelado de datos, con modelado de objetos usando varias notaciones (OMT, Booch, etc.), y algunas veces incluyendo técnicas adicionales de modelado de objetos como Casos de Uso y tarjetas CRC. La mayoría de estas organizaciones están adoptando e incorporando el UML como la notación orientada a objetos de sus metodologías.

Algunos modeladores usarán un subconjunto de UML para modelar solo lo que necesitan, por ejemplo simplemente el diagrama de clases, o solo los diagramas de clases y de secuencia con Casos de Uso. Otros usarán una suite más completa, incluyendo los diagramas de estado y actividad para modelar sistemas de tiempo real, y el diagrama de implementación para modelar sistemas distribuidos. Aun así, otros no estarán satisfechos con los diagramas ofrecidos por UML, y necesitarán extender UML con otros diagramas como modelos relacionales de datos y 'CRC cards'.

2.1. Casos de Uso

El objetivo en cualquier diseño de software es satisfacer los requisitos del usuario. Estos requisitos pueden ser requisitos de software, requisitos de productos, o requisitos de pruebas. La meta de capturar y comprobar los requisitos del usuario es asegurar que todos los requisitos sean completados por el diseño, y que el diseño sea acorde con los requisitos especificados.

Muchas veces los requisitos del sistema ya existen previamente en forma de documentos de requisitos. Los casos de uso se utilizan en esos casos para correlacionar cada escenario con los requisitos que completa. Si los requisitos no existen, modelar el sistema a través de los Casos de Uso, permite el descubrimiento de los requisitos.

En UML se utilizan los Casos de Uso para modelar los procesos de negocio.

El Caso de Uso da el punto de entrada para analizar los requisitos del sistema, y el problema que se necesita solucionar.

Los casos de uso se complementan muchas veces con los Diagramas de Actividad, que se pueden usar para modelar escenarios gráficamente.

Una vez que el comportamiento del sistema está captado de esta manera, los casos de uso se examinan y amplían para mostrar qué objetos se interrelacionan para que ocurra este comportamiento. Los Diagramas de Colaboración y de Secuencia se usan para mostrar las relaciones entre los objetos.

2.1.1. Definición de Caso de Uso

Un Caso de Uso es un contrato entre los stakeholders de un sistema y el comportamiento que el sistema debe ofrecer. [Cockburn]

Los casos de uso representan el comportamiento del sistema bajo varias condiciones, los escenarios que el sistema atraviesa en respuesta a los pedidos de uno de los stakeholders, llamado el actor principal.

Cada caso de uso se documenta como una descripción del escenario. La descripción puede ser escrita en modo de texto o como una secuencia de pasos ejecutados. Cada caso de uso puede ser también definido por otras propiedades, como las condiciones anteriores y posteriores del escenario, condiciones que existen antes de que el escenario comience, y condiciones que existen después de que el escenario se completa.

Los actores representan a los usuarios y otros sistemas que interaccionan con el sistema, pero no a las instancias en particular, sino a los tipos de usuario o roles.

El actor principal inicia una interacción con el sistema para completar algún objetivo. El sistema responde cuidando los intereses de todos los stakeholders. Diferentes secuencias de comportamiento, o escenarios se revelan en cada pedido y dependiendo de las condiciones de entorno. El caso de uso reúne todos esos diferentes escenarios.

Los Casos de Uso son fundamentalmente texto, aunque pueden escribirse usando diagramas de flujos, de secuencia, redes de Petri, o lenguajes de programación. En circunstancias normales, sirven para comunicarse de una persona a la otra, o a un grupo, y a veces sin entrenamiento especial. Texto simple es por eso usualmente la mejor opción.

El caso de uso se puede usar para discutir un nuevo sistema. Luego se puede usas para documentar los requerimientos actuales. Otro equipo podría luego documentar el diseño final, con el mismo formato.

Se podría hacer esto para el sistema de toda una compañía o para una simple parte de software. Lo bueno es que las mismas reglas básicas se aplican en estas distintas situaciones, aún cuando se haga con distinto rigor en detalles técnicos.

El actor principal es el que tiene el objetivo que el caso de uso debe resolver.

El flujo básico es el caso en el que nada va mal.

Las extensiones o flujos alternativos son lo que puede suceder diferentemente en un escenario.

Cada escenario muestra una secuencia diferente de interacciones entre actores y el sistema. [UML]

Los casos de uso son usados ampliamente porque describen mediante historias cómo el sistema se comporta. Los usuarios del sistema pueden ver lo que el sistema hará. Ellos pueden reaccionar tempranamente para refinar o rechazar las historias.

El primer momento que los casos de uso son valorables [Cockburn] es cuando son llamados como los objetivos de los usuarios y juntados en una lista. Esa lista de objetivos dice lo que el sistema hará. Revela el alcance del sistema, y se convierte en un medio de comunicación entre los diferentes stakeholders del proyecto.
Esa lista se examinará por usuarios representativos, desarrolladores expertos, y gerentes de proyecto. Estimarán el costo y la complejidad del nuevo sistema de esa lista. Negociarán sobre que funciones se realizarán primero, y como se dividirán los equipos.

El segundo momento que el caso de uso aporta valor es cuando sus redactores hacen un brain-storming de todas las cosas que pueden salir mal y las documentan como flujos alternativos.

2.1.2. Formatos de Casos de Uso

Los casos de uso pueden ser usados en diferentes situaciones:

- Un proceso de trabajo de negocio
- Para poner foco en un nuevo sistema a desarrollar que no tiene requerimientos aún
- Para ser los requerimientos funcionales para un sistema
- Para documentar el diseño de un sistema
- Pueden ser escritos por un grupo pequeño, o en grandes grupos distribuidos.

Cada situación requiere un estilo de escritura sutilmente diferente, y puede ser desde casos de uso casuales, hasta muy formales y completos.

Un ejemplo [Cockburn] de estilo formal es:

- Una columna de texto en los flujos, no una tabla.
- Sin sentencias condicionales, o sea solo oraciones afirmativas.
- Pasos numerados, combinándolos con letras para representar alternancia a flujos o subflujos. Ej.: 1.a.4 es el cuarto paso secuencial del flujo ‘a’, que es alternativo al primer paso del flujo principal.
- Contexto de uso
- Alcance
- Nivel
- Actor principal

- Stakeholders e intereses
- Garantías mínimas
- Garantías de éxito
- Disparador del flujo
- Flujo principal
- Flujos alternativos
- Validaciones
- Información extra

Un ejemplo de un estilo informal es representando los flujos como tablas de dos columnas en la primera las acciones de usuario y en la segunda las reacciones del sistema.

Los formatos de los casos de uso expresan básicamente la misma información [Cockburn], y por ende podemos encontrar un equilibrio con un formato que permita expresar las cosas cómodamente pero con el rigor adecuado.

2.1.3. Buenas Prácticas para redactar Casos de Uso

Un caso de uso bien escrito [Cockburn] es fácil de leer.
Consiste de oraciones escritas en una sola forma gramatical, con pasos de acciones simples, donde un actor obtiene un resultado o pasa información a otro actor.

Aprender a leer un caso de uso no debe tomar más de unos pocos minutos de entrenamiento.

Aprender a escribir buenos casos de uso es mucho más difícil. El escritor debe manejar tres conceptos para aplicar en cada oración del caso de uso, y usar al caso de uso como un todo. Estos conceptos son:

- Alcance: ¿Cuál es realmente el sistema a analizar?
- Actor primario: ¿Quién tiene el objetivo?
- Nivel: ¿Cuán alto o bajo es el nivel del objetivo?

La secuencia de pasos de los flujos puede ser desde pasos muy básicos hasta pasos compuestos que enuncian combinaciones de cosas que suceden conjuntamente según se requiera, o desee expresar.

El actor primario tiene la meta, el sistema debe ayudar al actor primario a alcanzar la meta. Algunos escenarios muestran como se alcanza la meta, otros terminan abandonando la meta. Cada escenario contiene una secuencia de pasos que muestran como las acciones e interacciones se revelan. Un caso de uso junta todos estos escenarios, mostrando todas las formas que la meta se alcanza o se falla.

* Algunos escenarios terminan con éxito, otros fallando al objetivo

* Un caso de uso contiene a todos los escenarios, los de éxito y los de falla.

* Cada escenario es una descripción para un conjunto de circunstancias con una salida.

Una acción es una de estas tres clases:

- Una interacción entre actores, en donde información puede ser pasada.
- Una validación, para proteger los intereses de algún stakeholder.

- Un cambio de estado interno, también para proteger o reforzar el interés de un stakeholder.

Un escenario consiste de pasos de acción. En un escenario exitoso, todos los intereses de los stakeholders se satisfacen. En un escenario de falla, los intereses de los stakeholders son protegidos de acuerdo a las garantías del sistema.

Los pasos bien escritos en los casos de uso están en una sola forma gramatical, son acciones simples en donde el actor cumple una tarea o pasa información a otro actor. Ej.:
El usuario ingresa su nombre y contraseña.
El sistema verifica que el nombre pertenece a un usuario válido y se corresponde con su contraseña.

Guía para redactar pasos de caso de uso correctamente:

1: Usar gramática simple: Sujeto….verbo…objeto directo.

2: Debe mostrar quien tiene la pelota (poner el actor principal o el sistema).
Ej. : El *usuario* ingresa….
Ej. : El sistema muestra…

3: Debe mostrar avance hacia el objetivo del caso de uso. Para esto no debe ser ni muy detallado ni demasiado corto. Un flujo básico debe tener 10 líneas como mucho, sino descomponer el caso de uso y o agrupar acciones que no produzcan reacciones en el sistema.
Ej. : “El usuario ingresa su nombre” + “El usuario ingresa su contraseña” reemplazar por “El usuario ingresa su nombre y contraseña”.

4: Debe mostrar la intención del usuario, no los movimientos en la interfaz. Ej.: Obviar pasos o partes del mismo que mencionen cosas cono: “… clickea en el

botón de…" ya que lo que nos interesa es la funcionalidad, y no la interfaz que puede cambiar independientemente de la necesidad funcional.

5: No debe chequear condiciones, las valida.
Ej.: Cambiar pasos como "… verifica si la contraseña es correcta y..."
por "…valida que la contraseña es correcta y … "

6: La condición dice lo que el sistema detecta, no lo que sucede.
Ej. : Cambiar pasos como "… el cliente olvidó su contraseña..."
por "…no se ingresa la contraseña… " ya que no importa si se la olvidó o le dio un paro cardíaco antes de ingresarla, lo que importa es el no ingreso.

7: Estadísticamente es más legible y entendible numerar los pasos.
Además si están numerados es útil para poder referenciar a otros pasos.

* Los flujos deben siempre tener postcondiciones. Si no hay postcondiciones significa que no se sabe realmente que hace un flujo y hay que relevar mejor la funcionalidad.

* Se recomienda una numeración de los pasos de los flujos que sea unívoca y auto-explicativa de su posición en el flujo.

Ej.: Números consecutivos para los pasos secuenciales y letras para los alternativos. Luego, 1.a.2.b.4 se sabe que es el cuarto paso de un flujo secuencial, que es a su vez el segundo alternativo del segundo paso secuencial de un flujo alternativo al primer paso del flujo básico
Ejemplo [Cockburn] de formato formal de Caso de Uso:

1. Nombre del Caso de Uso: Registración en cursos.

Breve descripción:

Este caso de uso permite al estudiante registrarse para cursos que se ofrecen en el actual semestre. El estudiante también puede modificar o borrar selecciones de cursos si los cambios son hechos al principio del semestre. El catálogo del sistema provee una lista de todos los cursos que se ofrecen en el semestre actual.

El actor principal de este caso de uso es el estudiante.

2. Flujo de eventos:

El caso de uso comienza cuando el estudiante selecciona “mantenimiento de cronograma” desde el formulario principal.

2.1. Flujo básico:

2.1.1. Crear un cronograma

2.1.1.1. El estudiante selecciona “crear cronograma”.

2.1.1.2. El sistema muestra un cronograma en blanco.

2.1.1.3. El sistema muestra una lista de todos los cursos disponibles.

2.1.1.4. El estudiante selecciona 4 cursos primarios y 2 secundarios y presiona submit.

2.1.1.5. Se ejecuta el subflujo “Agregar curso” para cada uno de los cursos agregados en el cronograma.

2.1.1.6. El sistema guarda el cronograma.

2.2. Flujos alternativos:

2.2.1. Modificar el cronograma

2.2.1.1. El estudiante selecciona “modificar cronograma”.

2.2.1.2. El sistema muestra el cronograma actual del alumno.

2.2.1.3. El sistema muestra una lista de todos los cursos disponibles.

2.2.1.4. El estudiante edita los cursos agregando o eliminando cursos, al final presiona submit.

2.2.1.5. Se ejecuta el subflujo “Agregar curso” para cada uno de los cursos agregados en el cronograma.

2.2.1.6. El sistema guarda el cronograma.

2.2.2. Eliminar el cronograma

2.2.2.1. El estudiante selecciona “eliminar cronograma”.

2.2.2.2. El sistema muestra el cronograma actual del alumno.

2.2.2.3. El estudiante presiona eliminar.

2.2.2.4. El sistema pregunta si esta seguro

2.2.2.5. El estudiante confirma la eliminación.

2.2.2.6. El sistema guarda la eliminación.

2.2.3. Guardar el cronograma

En cualquier parte del flujo el estudiante puede elegir guardar el cronograma seleccionando la opción “guardar cronograma”, guardándose los cursos seleccionados por el estudiante.

2.2.4. Agregar curso

El sistema se fija si el estudiante cumple los requisitos del curso y si el mismo esta abierto. En ese caso se marca al curso como registrado en el cronograma.

2.2.5. Requisitos incumplidos o curso lleno

Si el estudiante no cumple los requisitos para el curso o el mismo se encuentra lleno, se muestra un mensaje de error.

2.2.6. Cronograma no encontrado

Si cuando el estudiante selecciona editar o eliminar cronograma, y el sistema no lo encuentra, se muestra un mensaje de error.

3. Requerimientos especiales

   En este caso de uso no existen requerimientos especiales.

4. Precondiciones

   4.1. El estudiando debe estar logueado al sistema para comenzar este caso de uso.

5. Postcondiciones

   No hay post condiciones asociadas a este caso de uso.

6. Puntos de extensión

   No hay puntos de extensión asociadas a este caso de uso.

2.2. Casos de Pruebas

Los casos de uso se utilizan también para probar el sistema y ver si satisface los requisitos iniciales, ya que proveen una forma sencilla de testear la funcionalidad del sistema creando los casos de prueba de los mismos por los testers.

Los pasos de los casos de uso se van testeando para ver si el sistema está satisfaciendo los requisitos del usuario.

2.2.1. Definición de Caso de Prueba

Un Caso de Prueba es un documento con las pruebas para testear un software.

Se define en [Test] como una entrada y un resultado esperado. Básicamente se tiene por cada prueba: un evento disparador, una acción, una entrada, y un resultado esperado. Puede ser abstracta o detallarse hasta el nivel que se necesite. En general también se tiene una serie de pasos para poder ejecutar la prueba.
Campos opcionales son el ID del caso de prueba, el paso de prueba u orden de ejecución, requerimientos relacionados, la profundidad, categoría de la prueba, autor de la prueba, ejecutor de la prueba, e indicaciones si el caso de prueba es automatizable y si ya ha sido automatizado.

Los casos de prueba se utilizan para comparar los componentes de software contra los requerimientos de los cuales se trataron de satisfacer, para estipular si se cumplen, si se cumplen parcialmente o si no se cumplen.

Los Casos de Prueba pueden guardarse en un documento Word, una hoja de cálculo, una base de datos u otro repositorio.

2.2.2. Diferentes tipos de pruebas

Pruebas de Unidad (Unit Tests): testean a los componentes o módulos básicos de software, llamados unidades. Cada unidad es testeada para verificar que el diseño detallado para esa unidad se haya implementado correctamente.
Una unidad es la parte más pequeña testeable de una aplicación.
En programación procedural, una unidad puede ser un programa individual, función o procedimiento, mientras que en programación orientada a objetos, la unidad más básica es generalmente una clase, aunque a veces no interesa testear a un nivel tan bajo y se testean las funcionalidades cubiertas por un conjunto de clases.
Idealmente cada caso de prueba es independiente de los otros.
La meta de las pruebas de unidad es aislar cada parte del sistema y mostrar que las partes individuales son correctas. Una prueba de unidad es un contrato que las partes de código deben cumplir.
Pruebas de Integración: exponen los defectos en las interfaces e interacciones entre los componentes o módulos integrados.

El propósito de las pruebas de integración es detectar inconsistencias entre las unidades integradas.

Mientras se construyen los elementos de un sistema, se van integrando y se deben ir testeando hasta que el software funcione como un todo.

Pruebas de Sistema: testean un sistema integrado para verificar que se cumplan los requerimientos. Este tipo de testing es de caja negra ya que no interesa en este punto el diseño interno o la lógica del código.
Se toma a todos los componentes de software integrados y que pasaron las pruebas de integración.
Las pruebas de sistema se limitan a buscar defectos del ensamble de componentes y del sistema como un todo.

Se componen de pruebas funcionales y no funcionales.
Entre las pruebas no funcionales se encuentran las de performance, de carga, de estrés, de volumen, de aceptación Alfa y Beta, etc.

Pruebas de Regresión: son las pruebas que buscan errores de regresión, que ocurren cuando partes de software que ya funcionaban se refactorizan, o se quitan o mejoran partes de código. También aparecen errores de regresión con los cambios de requerimientos.
Entre los métodos para las pruebas de regresión se incluye la re ejecución de las pruebas ejecutadas previamente y chequeando donde las fallas previamente arregladas reaparecen.
Se suelen automatizar estos tipos de pruebas para ejecutarlas automáticamente en forma diaria o semanal.

Pruebas de humo: son las pruebas que buscan errores en las funcionalidades elementales del sistema en un nivel básico. Se utilizan en general antes de cada entrega de nueva funcionalidad o refactoreo de código a QA, y sirven para decidir si existe un nivel aceptable de estabilidad en lo entregado, para invertir tiempo en la ejecución de las pruebas unitarias, o rechazar la entrega para que se corrijan los errores cuanto antes por el grupo de desarrollo.

2.2.3. Buenas Prácticas para Casos de Pruebas

El **diseño combinacional de pruebas** reduce el costo del testing. Un plan de test de un mes se puede reducir a una semana. Se logra buena cobertura y detección de fallas. [Combinatorial]

Diseñar un plan de testing es difícil y caro. Puede llevar meses de trabajo duro. Un sistema de tamaño moderado de 100.000 líneas de código puede tener un enorme número de escenarios posibles de prueba. Los testers necesitan una metodología para elegir las pruebas más representativas. El proceso ISO 9000 da alguna ayuda, especificando que cada requerimiento en el documento de requerimientos debe ser testeado. Sin embargo, testeando requerimientos individualmente no garantiza que trabajarán bien todos juntos para entregar la funcionalidad deseada.

El método de diseño combinacional reduce el número de pruebas para chequear las funcionalidades del sistema.

Para diseñar un plan de testing, el tester identifica a las variables que determinarán los posibles escenarios para el sistema a probar. Ejemplos de estas variables son las entradas de usuario, eventos del sistema, parámetros de configuración del sistema, y otros eventos externos.

Por ejemplo, para testear una interfaz de usuario, las variables son los campos de la pantalla. Cada combinación de variables brinda un escenario diferente. Como hay muchas combinaciones de variables para testear todos los posibles escenarios, el tester debe usar alguna metodología para seleccionar unas pocas combinaciones representativas para testear.

Para el diseño combinacional, se deben generar pruebas que cubran las combinaciones de a pares, triples o N-arias de las variables de pruebas.

Campos que deberían tener los Casos de Pruebas:
Los Casos de Prueba deberían tener mínimamente los campos más representativos para ejecutar la prueba, y estos son : evento disparador, acción, variables, y resultado esperado.

3. Ventajas de generar automáticamente casos de prueba en forma configurable

- Cuanto más se automaticen las tareas repetitivas de testing (como la creación y mantenimiento de los casos de prueba), el tester se podrá concentrar en hacer pruebas más minuciosas y detectar mayor cantidad de errores, mejorando la calidad de los desarrollos.

- Automatizando se minimizan los costos del testing, en tiempo y esfuerzo tanto para el desarrollo inicial, como en las adaptaciones para las nuevas versiones y re-ejecuciones de testing de regresión. [UML-based]

- Como se reduce el costo de la generación, se pueden obtener los casos de prueba en forma temprana, y con ellos, los desarrolladores pueden construir pruebas automáticas con algún framework, como JUnit, e implementar además en forma mucho más sencilla y efectiva “Testing Oriented Programming”.

- La cantidad de alternativas y combinaciones en la cobertura del testing es mucho mayor de lo que podría ser con una cantidad fija de casos de prueba escritos a mano. [self-checking]

- Automatizando la generación de los casos de prueba, se asegura la eficiencia del testing, ya que se asegura la cobertura de los caminos importantes y atrapar los errores frecuentes, tarea que no es fácil realizarla manualmente, pues requiere experiencia y concentración.

- Como se puede configurar la generación de los casos de prueba, se puede aplicar la misma configuración para todos los casos de uso, y se puede asegurar de esta manera un nivel de calidad homogéneo en todas las funcionalidades de un proyecto.

- Como se puede configurar la generación de los casos de prueba, reajustando la configuración, se asegura la minuciosidad en las pruebas que requieren los módulos claves, o se disminuyen las pruebas en los casos de menos valor o riesgo si el tiempo apremia.

- Con la obtención de los casos de prueba en forma temprana, se puede analizar y especificar el lote de datos que el tester necesitará para realizar sus pruebas, ganando tiempo y mejorando las posibilidades de obtener el lote a tiempo, ya que esta es otra actividad olvidada en las planificaciones y a veces no se llega a testear debidamente por falta de datos.

- Se asegura de no trabajar con Casos de Prueba repetidos, ya que el sistema elimina las variaciones repetidas, cosa muy difícil de detectar si se procede manualmente con Casos de Prueba en número considerables. Estas situaciones se pueden dar con acciones compuestas anidadas en el Caso de Uso.

4. Objetivo de la tesis

El propósito de este trabajo es poder generar Casos de Prueba en una forma automática, controlable y clasificable para poder validar las funcionalidades descriptas en los Casos de Uso a un costo menor al que actualmente se aplica en Ingeniería de Software en muchos proyectos, y promover de esta manera la calidad en los procesos de software.

Específicamente, se busca auxiliar al tester en sus tareas, ya que como se dijo antes, en muchos casos no se contempla al testing como actividad esencial, y por lo tanto no se planifican sus actividades en los proyectos (no se reservan tiempos para testing), conformándose luego con hacer un *testing de emergencia* para salvar la mayor cantidad de bugs para la entrega del producto, días y a veces horas antes de la entrega.

Otras veces, si se planifica el testing, se lo hace mal, y se subplanifica, dejando poco tiempo para las tareas necesarias, y como algunas tareas de testing demandan tiempo, se suele caer en el abandono de esta tarea, o su escasa aplicación en proyectos, lo que los llevará indefectiblemente a una baja en la calidad.

Otra causa es el difícil mantenimiento de los Casos de Prueba, que por su magnitud y modificaciones surgidas por cambios en los requerimientos, hace que esta tarea sea tediosa y muy repetitiva. [Rosetta]

Maximizar el rendimiento del testing es vital para aumentar la calidad de los productos.

La mejor forma de eficientizar la tarea de testing es automatizando los procesos involucrados que puedan ser automatizados, como la construcción de los casos de

prueba, la regeneración de los mismos con diferentes parámetros y automatizar las ejecuciones de los tests.

Automatizar el testing es claramente atractivo, pues la computación se expande cada vez mas en todos los aspectos, haciendo que los diseños sean más complejos y las interacciones entre componentes sean difíciles de medir y predecir. El testing del software se ha vuelto más difícil y más caro. [UML-automatic]

Para esto último, se cuentan con herramientas que memorizan las acciones del tester en las aplicaciones (llamadas robots) y que luego pueden ser parametrizadas y ejecutadas infinidad de veces.
El problema es que estas herramientas suelen ser caras, y requieren un perfil técnico del tester. Además se justifican si el volumen de repeticiones es elevado, ya que el ajuste de estas herramientas a veces lleva bastante tiempo.

El otro aspecto a resolver es crear las variaciones de pruebas para guiar al robot la primera vez, o ejecutarlas manualmente por el tester, y poder regenerar las pruebas con diversos criterios de granularidad según el tipo de prueba que se esté realizando. Este proceso suele llevar mucho tiempo ya que se confeccionan, corrigen y regeneran manualmente.

El alcance de pruebas elegido en este trabajo es el de ‘Pruebas de Unidad’, ya que es el más significativo de las funcionalidades y el más utilizado en el testing en general.

5. La Propuesta: Presentación de un aplicativo prototipo

Se propone la presentación de un aplicativo prototipo para crear variaciones de Casos de Prueba que sirva para mostrar prácticamente las conclusiones analizadas de las necesidades relevadas para el testing y que sirva asimismo como un ejemplo de las características mínimas que deberían tener las herramientas que se construyan para este fin, destacando la posibilidad de cambiar las configuraciones en la profundidad de las variaciones en sus dominios, en los caminos elegidos para constituir las pruebas, en las variaciones compuestas OR, y en el análisis de post-condiciones.

La posibilidad de diferentes configuraciones, cubrirá las necesidades para cada tipo de prueba.

La generación será combinacional (se generarán valores para cada variable conjuntamente) para reducir la cantidad de pruebas.

La salida debe ser en un formato legible para humanos, para que pueda ser utilizado como base y auxiliar la tarea del testing humano y asimismo la base para la creación posterior de las pruebas de unidad por desarrolladores, contribuyéndose a la calidad en los procesos de IS.

6. Desarrollo de la propuesta

6.1. Metodología de trabajo utilizada en la tesis

Al ser una investigación operativa con un resultado práctico implementado como un prototipo ejecutable, el desarrollo de la solución se plantea como cualquier desarrollo de software, en etapas que van desde el análisis a los resultados implementados: Análisis, Diseño, Implementación y Resultados.

6.1.1. Análisis

Para ayudar al tester a automatizar la construcción de los casos de prueba, se plantean dos procesos consecutivos:

El primero es obtener una representación de los Casos de Uso desde la descripción textual del usuario, a una estructura intermedia, para poder ingresarla al segundo proceso. Esta estructura intermedia es una forma de persistir los Casos de Uso, y se hará en un formato standard de intercambio de datos, y permita que cualquier otra aplicación pueda levantar los Casos de Uso.

El segundo proceso es la generación de los Casos de Prueba. Este proceso será parametrizable en los siguientes aspectos útiles para generar la cantidad y granularidad de los casos de pruebas para distintos tipos de pruebas y para distintas políticas de QA.

Análisis de los Casos de Uso

Como la descripción de funcionalidades de un sistema mediante Casos de Uso no está rigurosamente definida en UML, se suele usar lenguaje coloquial, o a veces demasiado orientado al dominio, pero con las ambigüedades propias del lenguaje libre que lo hacen poco preciso para su automatización.

Se define mas adelante una gramática con la suficiente representación textual de los Casos de Uso, pero con la formalidad requerida para el procesamiento de los mismos.

Como el alcance de este trabajo se centra en las llamadas pruebas de unidad que son las más habituales e importantes en el testing, no se manejan las extensiones ni las inclusiones de UML, dejándose los mismos notados como acciones textuales. Por ejemplo: 'includes: ir al CU xxx'.

Solo se debe testear lo explícito en los flujos, ya que se tiene un resultado esperado real, sino no tiene sentido testearlo, por eso no se contemplan los casos opuestos a lo definido en los flujos, ya que si un flujo valida por ejemplo un valor de un objeto, no significa que el flujo alternativo abarque todo lo restante. Se debe especificar con un flujo de Caso de Uso todo lo que se desea abarcar.

Se manejarán para las variables de pruebas de dominios standard y también definidos por el usuario.

Se manejarán rangos de los dominios para mayor expresividad y reflejo de la realidad.

Los flujos alternativos pueden ser tanto de éxito o de falla, pues si existen dos escenarios de éxito, y como solo uno puede ser el básico, por ende todos los otros flujos serán redactados como alternativos.

Los flujos alternativos se diferencian conceptualmente en comunes y “de error”. Los flujos alternativos de error al final del mismo retornan al mismo punto del que partieron, y son totalmente antagónicos al mismo. Esta separación es documentativa, aunque incide en la generación de los casos de prueba, ya que la descripción de los flujos alternativos comunes no se tiene en cuenta en los pasos de prueba porque son redundantes, distinto el caso de los flujos de error en donde el texto pone de manifiesto el antagonismo.

Luego de un paso de SALTO o de FIN de flujo, no se deben poner mas pasos en el mismo flujo, pues nunca se podrán acceder secuencialmente a estos.

Todo paso de FIN de Caso de Uso debe tener por lo menos una POST CONDICION.

Todo flujo se considera correcto si termina en un FIN de Caso de Uso o un paso de SALTO.

El flujo básico, debe terminar siempre en un paso de FIN de Caso de Uso.

Los pasos de SALTO deben ser a pasos existentes.

Los identificadores de cada paso deben ser unívocos: Se adopta la nomenclatura de [Cockburn] con números para indicar secuencia y letras para indicar alternancia (step-id = numero | numero.letra | numero.letra.step-id).

Dependencia / Independencia entre variables de prueba: Se asume que las variables son independientes por defecto, y para indicar dependencia entre variables se lo debe explicitar, explicitando para un valor de un dominio que valores del otro subdominio de la otra variable le corresponde.

Ejemplo: Para explicitar dependencia de una variable provincia a otra variable país, se crea un subrango de provincias por cada país, y se utiliza el adecuado en la acción del caso de uso.

Se testearán las variables de pruebas en una aproximación combinacional, ya que se asegura de esta forma una cobertura total de las variables en menos cantidad de pruebas, y se logra con esto una mayor eficiencia del testing.

El prototipo deberá asegurar una sintaxis adecuada, ya que los Casos de Uso y todas sus partes y subpartes se generarán por el usuario con la misma.

El prototipo deberá visualizar un árbol de flujos secuenciales y alternativos, ya que es la visualización más común para Casos de Uso.

Gramática simplificada de Casos de Uso

Como el objetivo de este trabajo es generar Casos de Pruebas, solo se modelan y utilizan las partes útiles a tal fin de los Casos de Uso, resultando en un modelo de Caso de Uso mas completo que el básico, pero menos que el mas exhaustivo que se pueda hallar, consiguiéndose de esta forma una mayor generalidad. A continuación se presenta la gramática conceptualmente establecida para este trabajo en Extended Backus Naur Form, [EBNF]:

```
(* Los siguentes son  los tipos básicos *)
value = string ;
name = string ;
step-id = string ;
version = string ;
comment = string ;
description = string ;
post-condition = string ;

char-beginning = char ;
char-ending =  char ;

float-beginning = float ;
float-ending = float ;

integer-beginning = integer ;
integer-ending = integer ;

date-beginning = date ;
date-ending = date ;
```

```
(* Un actor puede ser un usuario o el sistema *)
actor =
        name ,
        (
        user-actor |
        system-actor
        ) ;

(* Todas las comparaciones posibles para valores o rangos de valores*)
comparation =
        equal-comparation |
        distinct-comparation |
        great-comparation |
        great-or-equal-comparation |
        less-comparation |
        less-or-equal-comparation |
        in-comparation |
        not-in-comparation
        ;

basic-operand = value ;

(* Un rango de dominio es en general un subconjunto de un dominio mayor
y se usa como un dominio mas *)
range-domain =
        character-range-domain |
        date-range-domain |
        float-range-domain |
        integer-range-domain |
        user-defined-domain
        ;
```

```
(* Un operando, que es parte de cualquier acción, es simplemente un valor
o es un rango de dominio*)
operand =
        basic-operand |
        range-domain
        ;


(* Los dominios enumerativos poseen una lista con los valores que los
conforman *)
user-defined-domain  =  {value}+ ;

float-range-domain =
        float-beginning ,
        float-ending
        ;
integer-range-domain =
        integer-beginning ,
        integer-ending
        ;

character-range-domain =
        char-beginning ,
        char-ending
        ;

date-range-domain =
        date-beginning ,
        date-ending
        ;
```

```
domain =
        name ,
        (
        boolen-domain |
        character-range-domain |
        date-range-domain |
        float-domain |
        float-range-domain |
        integer-domain |
        integer-range-domain |
        user-defined-domain
        ) ;


variable =
        description ,
        domain ,
        ;

(*Todas las acciones poseen descripción, una comparación y un operando*)
action-common-body =
        description ,
        comparation ,
        operand
        ;

(* Solo las acciones simples poseen variables de prueba *)
simple-action =
        action-common-body ,
        variable
        ;
```

```
(* Las acciones compuestas tienen una lista con dos o más acciones *)
composed-action =
        action-common-body ,
        action ,
        {action}+
        ;

(* Una acción AND es una acción compuesta *)
and-action = composed-action ;

(* Una acción OR es una acción compuesta *)
or-action = composed-action ;

(* Las acciones son simples o compuestas *)
action =
        simple-action |
        composed-action
        ;



 (* Los pasos de salto pueden apuntar solo a pasos secuenciales,
alternativos o de fin de flujo *)
going-to-step =
        sequential-flow-step |
        alternative-flow-step |
        end-flow-step
        ;
```

```
(* A los flujos basicos o alternativos solo pueden seguirles pasos
secuenciales, que para emprolijar se pone el nombre de next-sequential-
step *)
next-sequential-step = sequential-flow-step ;



(* A los pasos secuenciales solo pueden seguirles mas pasos secuenciales,
pasos de fin o de salto*)
next-sequential-or-end-or-goto-flow-step =
        sequential-flow-step |
        end-flow-step |
        goto-flow-step
         ;


basic-flow =  next-sequential-step ;

(* El paso secuencial es el único con alternativos, actor y acción.
Representa a las oraciones del Caso de Uso. Los demas tipos de paso
salvo el paso de fin, son para control de flujo *)
sequential-flow-step =
        step-id ,
        comment ,
        actor ,
        action ,
        { alternative-flow-step }* ,
        next-sequential-or-end-or-goto-flow-step
        ;
```

```
(* El paso alternativo siempre tiene como posterior a un paso secuencial *)
alternative-flow-step =
        step-id ,
        isErrorAlternativeFlow,
        comment ,
        next-sequential-step
        ;

(* El paso de fin de flujo es el único con post-condiciones y sin paso
siguiente *)
end-flow-step =
        step-id ,
        {post-condition}+
        ;

(* El paso que le sigue a un paso de salto se llama going-to-step *)
goto-flow-step =
        step-id ,
        going-to-step
        ;

(* Un caso de uso tiene nombre, versión y un flujo básico *)
use-case =
        name ,
        version ,
        basic-flow
        ;
```

Análisis de los Casos de Prueba

El resultado de la generación de las pruebas debe constituirse en una tabulación de campos, siendo cada línea un test individual, siendo cada columna como sigue:

- **Pasos para reproducir el Test.**
- **Pasos recorridos del flujo del Caso de Uso.**
- **Variables (una columna por cada variable que se encuentre en el Caso de Uso).**
- **Resultado esperado.**

El evento disparador es el primer paso para reproducir el caso de prueba.

Los pasos recorridos del caso de uso son útiles para repasar y clasificar los casos de prueba con los flujos del caso de uso.

Los otros campos que se necesiten, como el identificador del caso de prueba, comentarios, etc. no se generan ya que son fácilmente agregados por el tester cuando los necesite.

Configuración de la generación de las pruebas

Se puede configurar en cuatro aspectos, el camino que se va recorriendo para ir generando las pruebas, los valores que se generan para cada variable, la cantidad de subacciones de una acción OR, y si se contemplan individualmente o no las post-condiciones del flujo.

Configuración del camino de pruebas: forma de recorrer los caminos del caso de uso para recolectar las pruebas. Se muestran en la Figura 1. y puede ser:

- **Camino básico** del caso de uso.
  Útil para pruebas de humo básicas.

- **Camino básico y los flujos alternativos** sin utilizar los pasos de salto.
  Útil para pruebas de humo más exigentes y pruebas unitarias básicas.

- **Camino mínimo:** Camino Básico y los flujos alternativos, utilizando los pasos de salto pero pasando una única vez por cada salto.
  Útil para pruebas unitarias en general.

- **Camino Exhaustivo:** ídem Varios Ciclos pero contemplando 2 pasadas por cada ciclo.
  Útil para pruebas unitarias y pruebas de regresión exigentes.

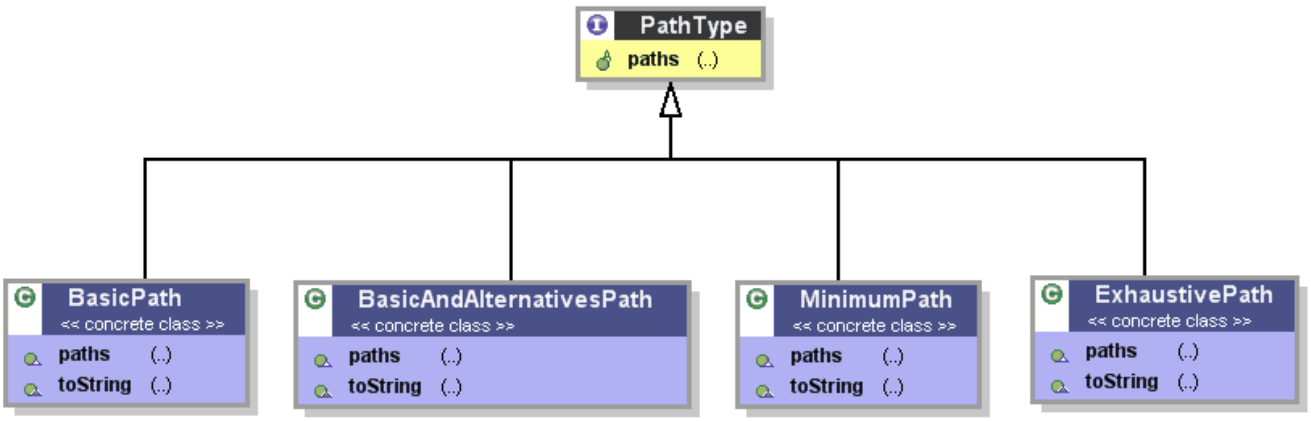

Figura 1. Configuración del camino de pruebas

Configuración del tipo de cobertura de los dominios de las variables de prueba: cantidad de variaciones por cada variable de prueba. Se muestra en la Figura.2 y pueden ser:

- **Mínima:** valor que cumple la acción o acciones en la cual la variable esta incluida.

- **Promedio:** valores límites del rango o de los subrangos del dominio de la variable que cumplen la acción o acciones en la cual la variable está incluida.

- **Exhaustiva:** en dominios chicos como los enumerativos o rangos cortos de enteros, todos los valores, sino una cantidad significativa de valores aleatorios además de los valores extremos.

Por ejemplo, si se tiene una variable entera con una condición 1<=x<=10, con una variación de prueba mínima se generará el valor 1 o 10 solamente, con una promedio se generará el 1 y el 10, y con una exhaustiva se generará el 1, el 10 y valores al azar entre el 2 y el 9.

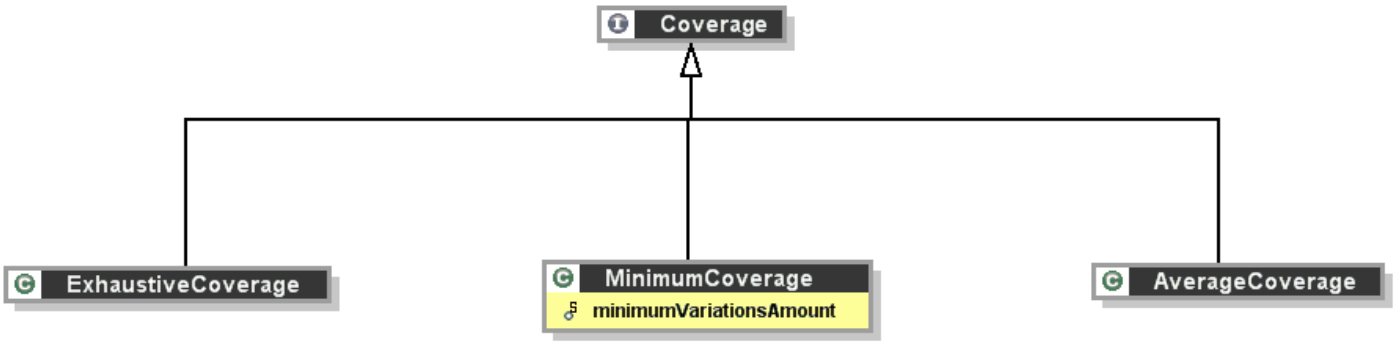

Figura 2. Tipo de cobertura para los dominios

Configuración de variaciones compuestas OR: forma de variar las subvariables que conforman una acción compuesta OR lógica. Se muestra en la Figura.3 y puede ser:

- **Solo una variable** elegida aleatoriamente de las que conforman una acción compuesta OR.

- **Algunas promedio** elegidas aleatoriamente de las que conforman una acción compuesta OR.

- **Todas las variables** que conforman una acción compuesta OR.

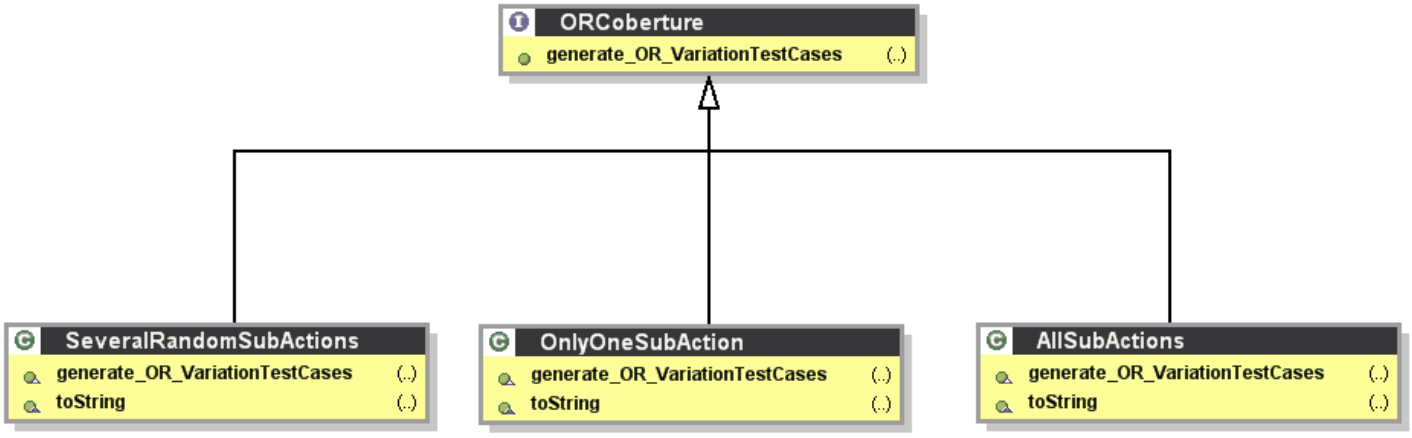

Figura 3. Configuración de variaciones compuestas OR

Configuración de las post condiciones: forma de variar las post condiciones de los flujos. Se muestra en la Figura.4, y puede ser:

- **Todas juntas:** las postcondiciones de un flujo se testean en un solo test.

- **Cada una separadamente:** las postcondiciones de un flujo se testean en forma individual, un test para cada una.

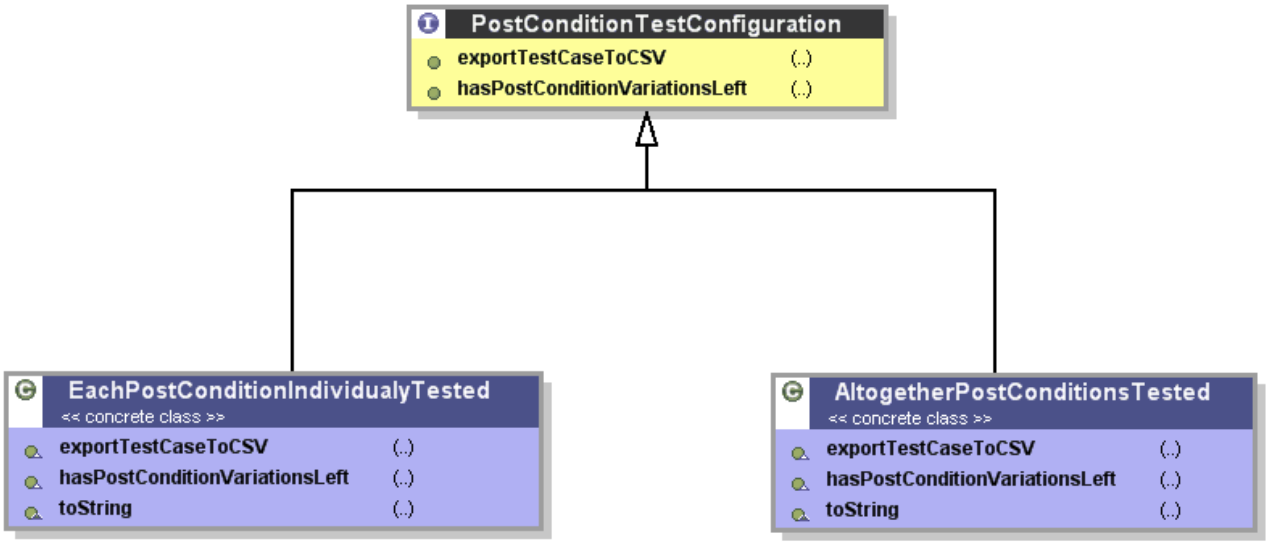

Figura 4. Configuración de las post condiciones

6.1.2. Diseño

Diseño de los Casos de Uso

El diseño de los casos de uso se realizó implemetando la gramática simplificada de los casos de uso desarrollada en el punto 6.1.1, la cual se generó contemplando conceptos generales de casos de uso, para poder expresar cualquier situación modelable con casos de uso.

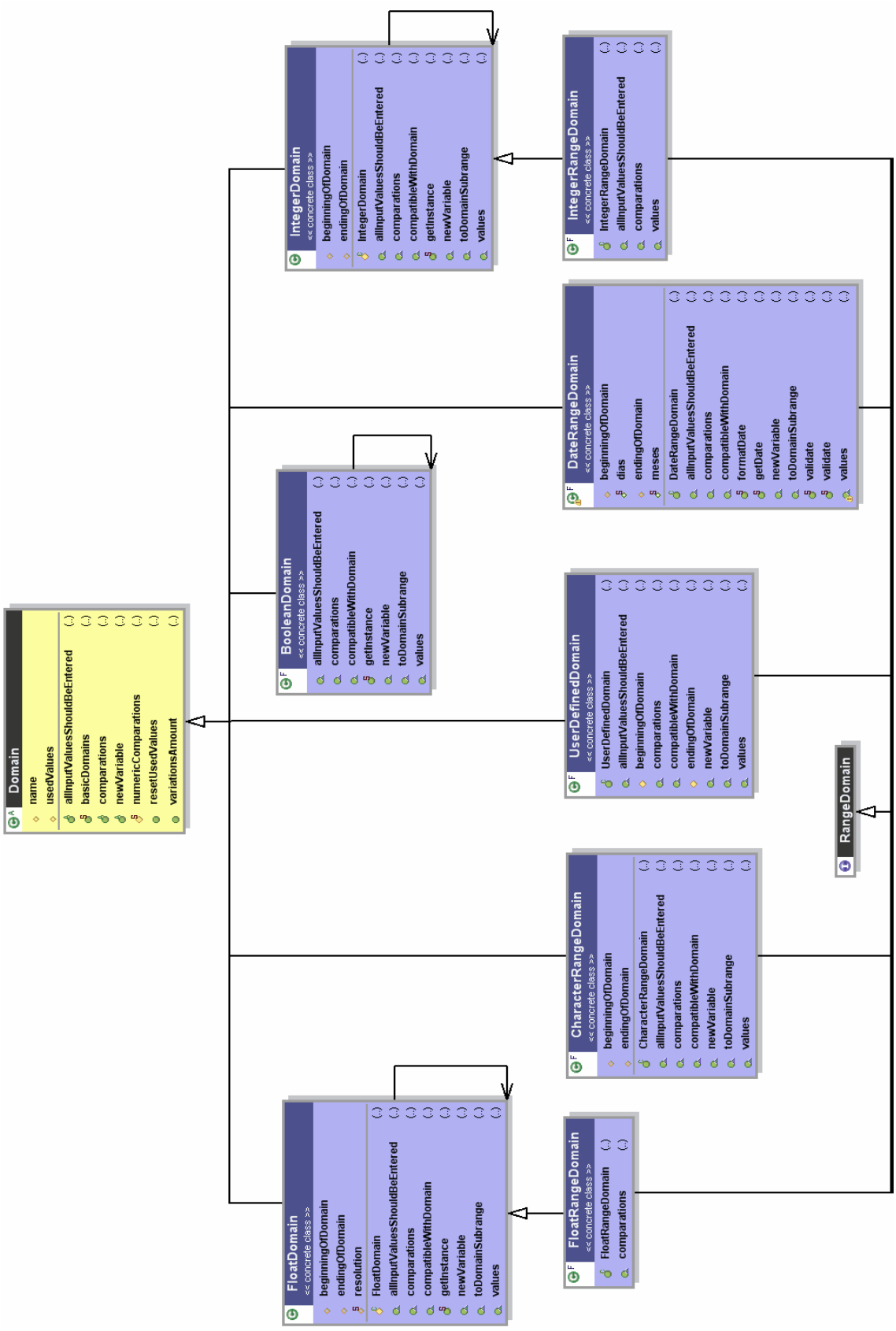

Figura 5. Dominios de las variables de prueba

En la figura 5 se muestran los dominios contemplados, con los cuales se pueden modelar todas las situaciones.

Se utilizan rangos de dominio para modelar los subconjuntos de dominios que se necesitan tratar como dominios. Por ejemplo si el dominio de una variable de prueba son los enteros entre el 1 y el 10, se crea un dominio que sea un rango de enteros.

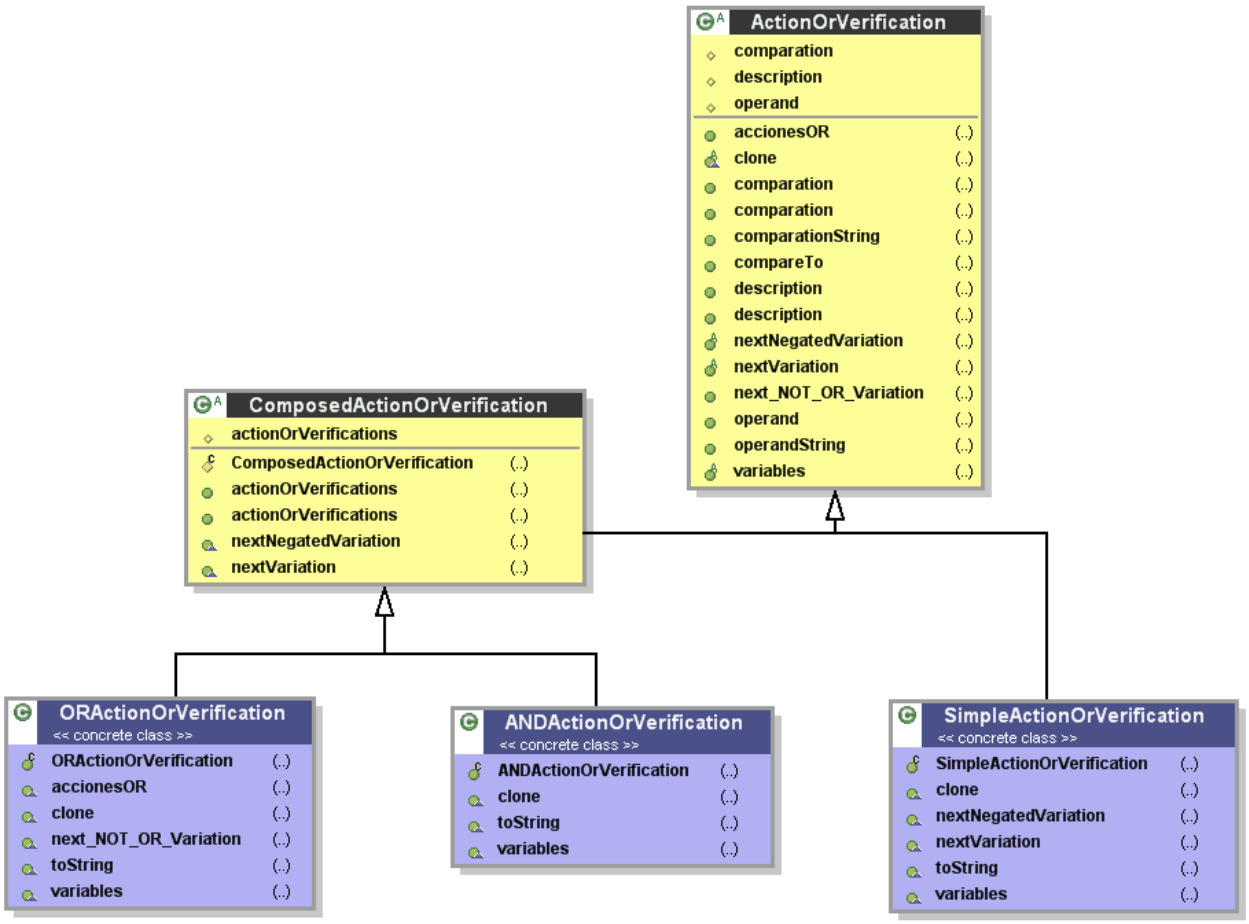

Figura 6. Tipo de acciones

Solo las acciones simples tienen variables de prueba. Las acciones compuestas tienen una lista con dos o más acciones, tanto simples como compuestas y obtienen las variables recorriendo sus sub acciones. Figura.6.

Los pasos de los flujos de los casos de uso se manejan como listas para la mayoría de los tipos de pasos (Figura.7), salvo el paso de fin de flujo que no tiene siguiente.
Según cada tipo, pueden tener a ciertos tipos de pasos siguientes, lo cuál está detallado en la gramática del punto 6.1.1.

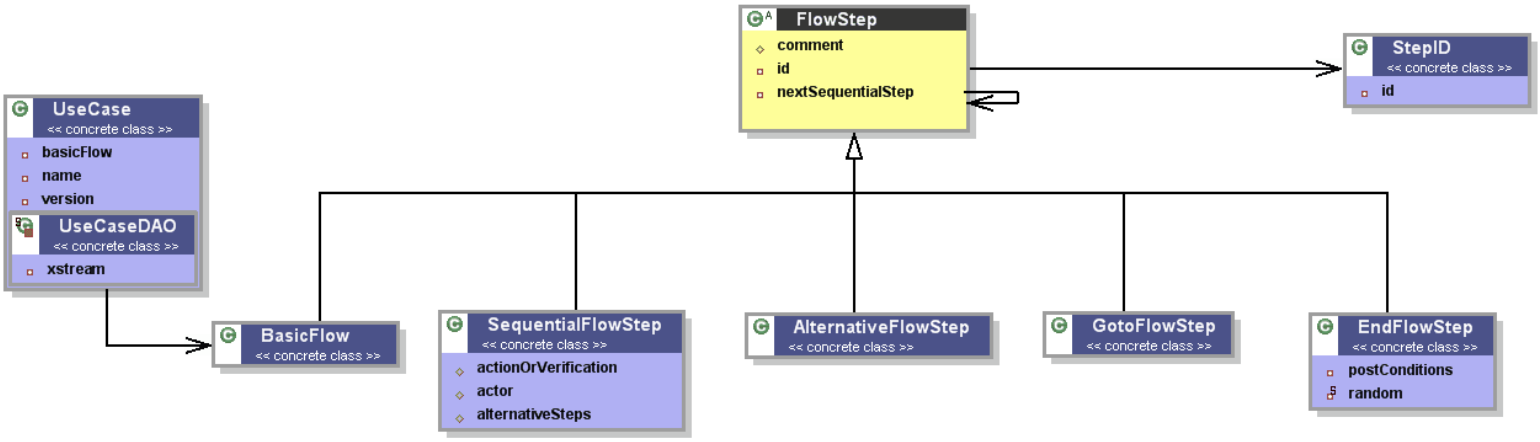

Figura 7. Componentes de los flujos de casos de uso

Solo los pasos secuenciales poseen actor y acción, pues son los que conforman los flujos de acciones del caso de uso, el resto de los pasos son modificadores de estos flujos, pues los hacen alternativos (AlternativeFlowStep) o los finalizan (EndFlowStep) o los reutilizan saltando a flujos anteriores (GotoFlowStep).

Como existen flujos alternativos “de error”, donde al final del mismo retornan al mismo punto del que partieron, y son totalmente antagónicos al mismo, si se recorriesen los mismos en un camino de prueba, la variable del paso de origen no tendría valores posibles, ya que debería pertenecer a dos conjuntos disjuntos. Por ejemplo: pago > 0 (ok) y pago <= 0 (error).
Para resolver este problema, se eligió la posibilidad de marcar en la creación del flujo alternativo con un flag de error, y si el mismo es de error, se muestra como un paso de prueba, pero no se contempla para reducir el dominio de prueba de la

variable de prueba en cuestión. Siguiendo el ejemplo anterior, no se tomaría el <= 0 para la generación de valores que aparecerán en la columna de la variable en el archivo de salida.

Las comparaciones que se pueden aplicar a las variables de prueba para formar las acciones de los pasos de caso de uso se detallan en la Figura 8.

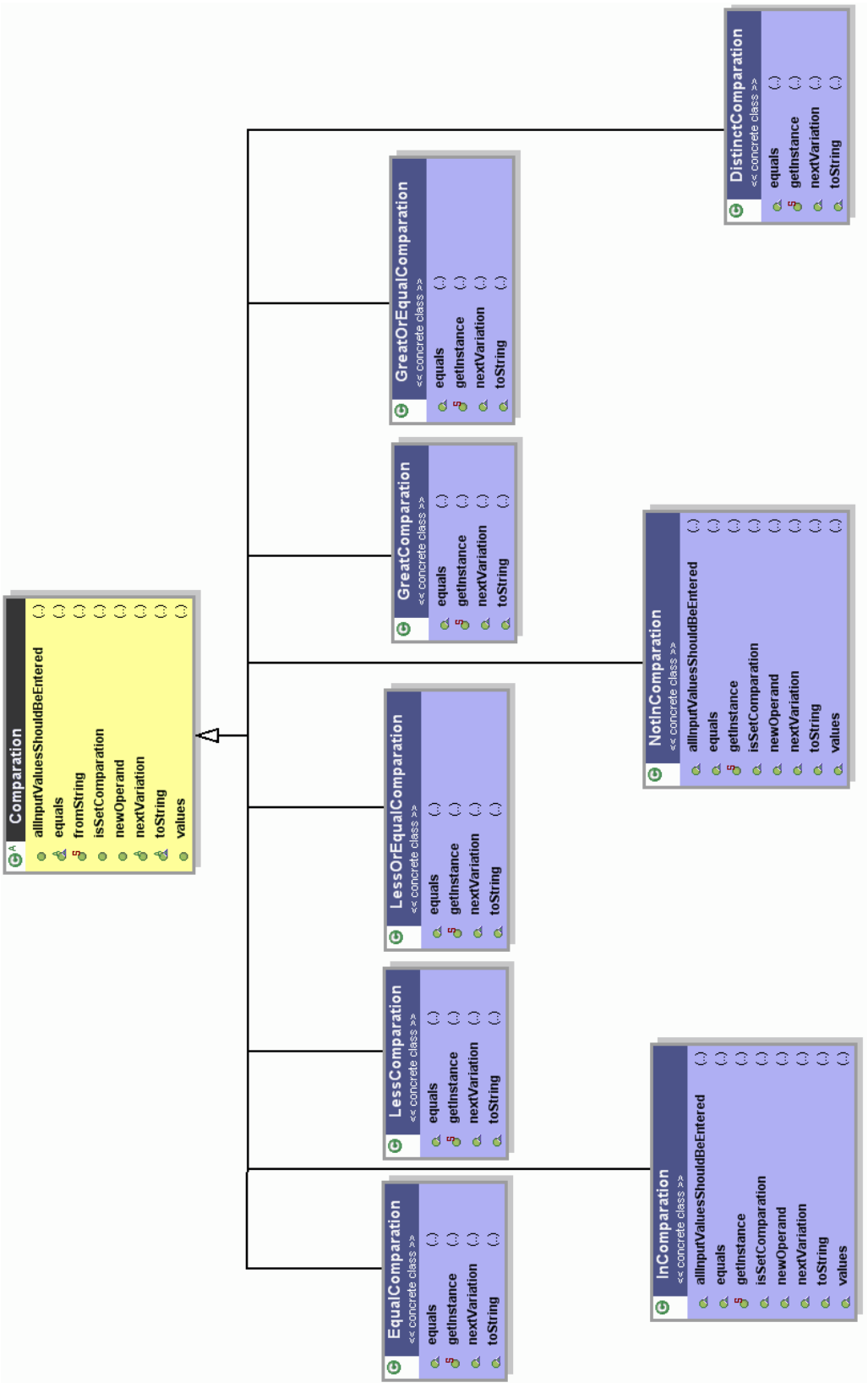

Figura 8. Comparaciones de variables

Se modeló a los actores como usuarios, a los cuales se le dan nombres, y al sistema, como un usuario más Figura.9.

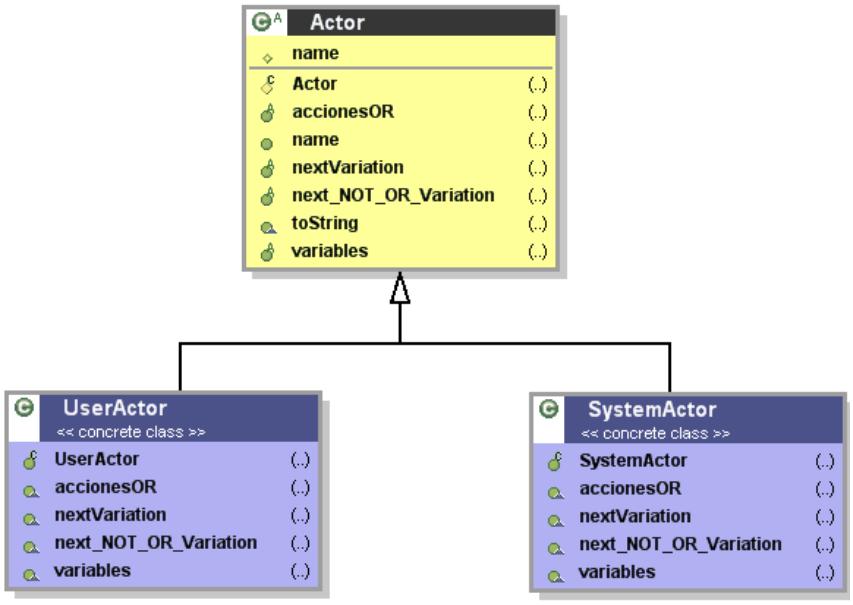

Figura 9. Tipos de actores

Diseño de los Casos de Prueba

Como la generación de los casos de prueba se realiza recorriendo los flujos de los casos de uso y generando automáticamente una salida a archivo, el diseño de los casos de prueba carece de estructura como la de los casos de uso, y consiste en un conjunto de clases que modelan el algoritmo.

Se mantiene en cada dominio de variable de prueba (Figura.10) los valores que se van utilizando en la generación de pruebas, así no se repiten en lo posible y se obtiene una mayor cobertura. Luego de la generación, a los dominios de variables se les resetean estos valores.

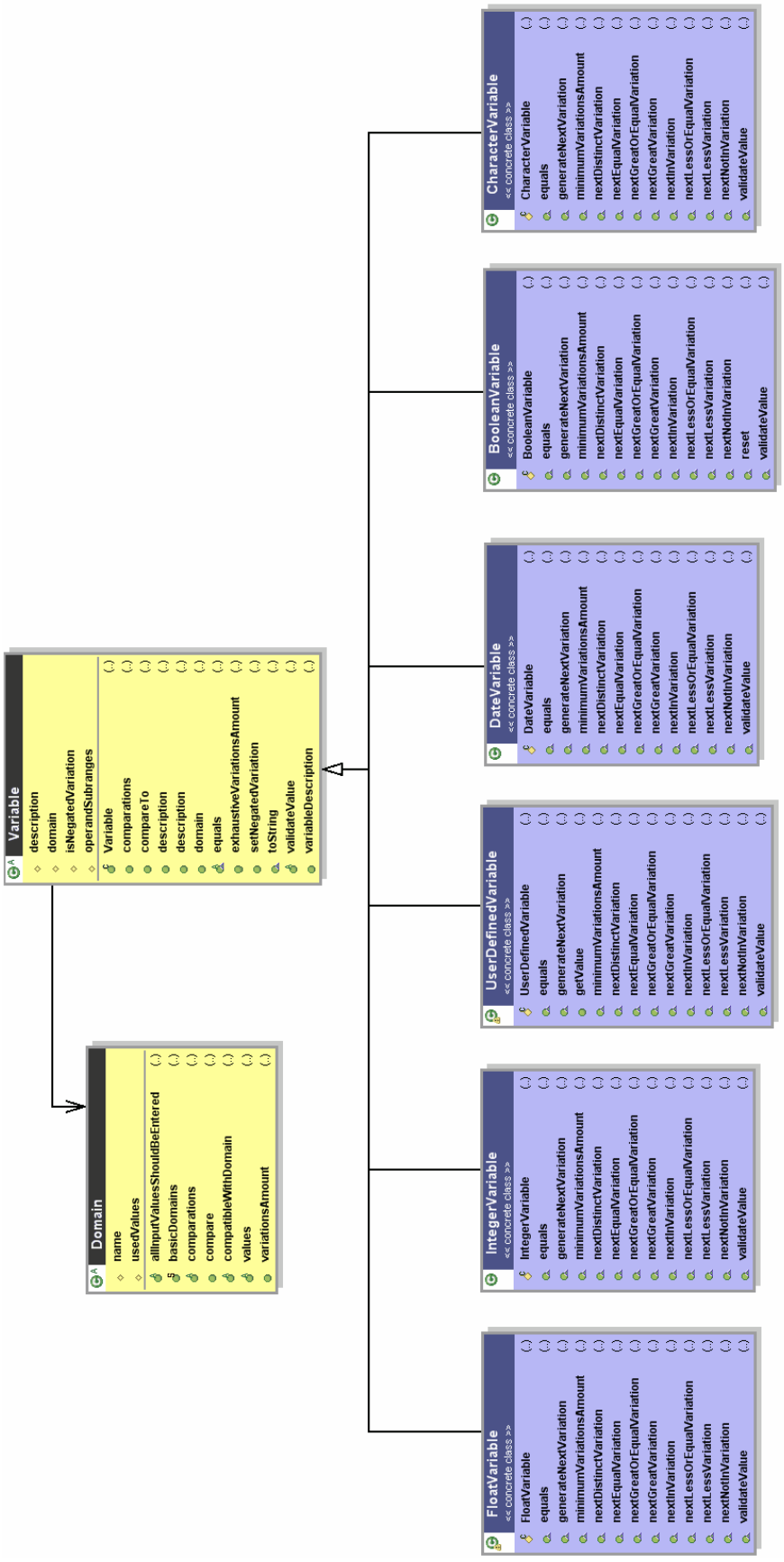

Figura 10. Tipos de variables de prueba

Dado que para ir cumpliendo las diferentes condiciones de las acciones en los pasos del caso de uso que se van recorriendo para generar las pruebas se pueden encontrar mas de una vez una misma variable, se pueden dar situaciones como agujeros en los dominios de las mismas con valores que no pueden tomar dichas variables para mantener verdaderas las acciones contenidas en un paso de prueba. Para modelar esto, se tienen los llamados “Subrangos de Dominio”, que son los subrangos de valores permitidos, y se muestra en la Figura. 11.

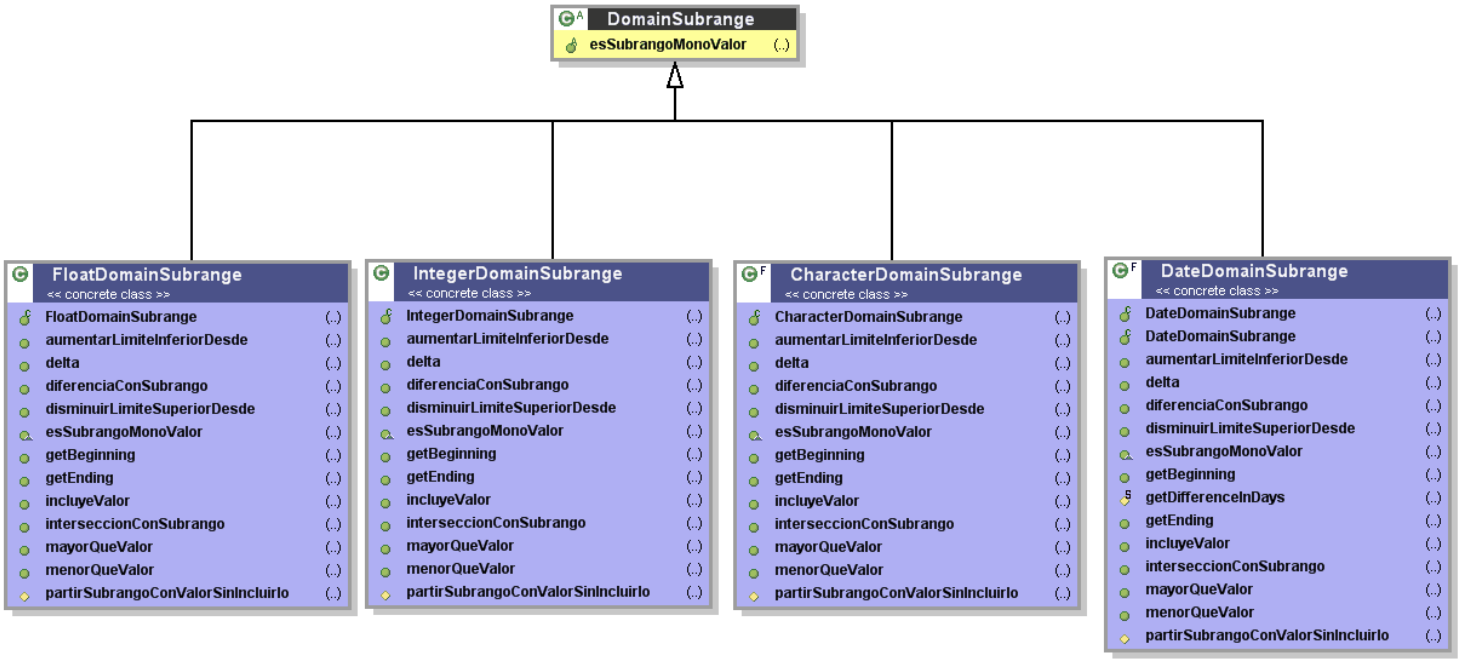

Figura 11. Subrangos de dominio de variables

Los subrangos booleanos no necesitan los subrangos de dominio por su trivialidad.
Los subrangos de dominios definidos por el usuario se modelan utilizando los subrangos de enteros como índices a los valores del dominio original.

Cuando se están generando variaciones en una acción compuesta OR en forma individual, se necesitan variaciones negativas, o sea que sean falsas, y esto se

logra seteando un flag de “negada” en las variables. Este flag es obviamente reseteado con cada camino de prueba.

Decisiones de diseño

* Se eligió para la persistencia de los objetos del contexto a [XStream] por razones de facilidad de uso, porque es transparente para el modelo de clases de negocio, porque no necesita archivos de mapeos, ni emplear un compilador de esquema o [DTD], pues serializa usando la API de reflection de Java.

El enfoque de serialización de Xstream es mucho mejor que el binding Java-XML, pues para hacer el binding se necesita una etapa de preproceso que genere el código de un esquema XML o DTD. Para la serialización se utiliza una conversión en tiempo de ejecución que utiliza la API de reflection. La serialización provoca menos esfuerzo pues no necesita generación de código fuente ni pasos de compilación. El único inconvenientes la necesidad de tener la misma librería para la serialización y para la deserealización.

* Se eligió implementar en el aplicativo el parseo de casos de uso en XML, para que la importación y exportación de los casos de uso se hagan sin la utilización de alguna biblioteca como [XStream], para evitar la dependencia de la misma en la generación de los casos de uso, posibilitandose de esta forma la importación de casos de uso desde archivos XML generados por cualquier editor o herramienta, con el solo requisito de cumplir un DTD creado con la gramatica de casos de uso del punto 6.1.1. El mismo se expone en el punto 6.1.3.

* Se descartó la implementación del prototipo como un pluggin de Eclipse [emf] o integración con alguna otra herramienta existente [ePlatero] pues no quería perderse generalidad y la posibilidad de ejecutar el prototipo en cualquier máquina con el solo requisito de tener la máquina virtual de Java instalada, ya que muchas

veces las máquinas destinadas a testing carecen de recursos para correr debidamente entornos de desarrollo como Eclipse.

* Se contemplaron algunos editores de casos de uso freeware como [SUMR], por su claridad y facilidad visual.

* Se descartó utilizar JavaCC [JavaCC], [JavaCC-2] para la construcción del prototipo pues la complejidad de la estructura de los casos de uso modelada no es tan compleja para requerirlo, y la mayor riqueza está en el comportamiento del generador más que en la estructura de los casos de uso.

* Como la generación de los casos de prueba debe ser en un lenguaje neutral y fácil de editar y agregar contenido para humanos, se eligió que se generen en formato de texto separado por comas (csv) por su fácil importación en herramientas comunes como MS-Excel. Se eligió un formato representativo y simple, parecido al utilizado en [UC-test], que contiene la información necesaria para utilizar en los diferentes casos de prueba.
* Debido a que la definición de [Cockburn] de que las acciones de los flujos de los casos de uso son simples no es un estándar en muchos sitios, se eligió contemplar condiciones compuestas, como las lógicas AND y OR, necesitando estas últimas una contemplación especial para la generación de pruebas, y por esto se incluyó entre los elementos configurables.

* Se generan (en lo posible) valores nuevos para las variables de pruebas en el contexto del caso de uso, o sea que si el dominio de una variable tiene la suficiente cantidad de valores, se generarán valores distintos que cumplan las condiciones del caso de prueba para esa variable, primero cubriendo los valores extremos, y luego se toman valores al azar, asegurándose una mejor cobertura y posibilidad de atrapar mas errores con las combinaciones de valores. [Combinatorial]

6.1.3. Implementación

La implementación se realiza en Java 1.4 por ser un standard hoy en día.

Se utilizó Java Swing para la parte gráfica, ya que no es necesario nada más que tener una JRE instalada en la máquina a utilizar.

El prototipo se empaquetó en un jar, y se entrega en un archivo comprimido (.zip) junto con dos jars de xstream y un .bat para correr el prototipo desde un entorno Windows fácilmente con el unico requisito de tener java 1.4 o superior.

6.1.4. Resultados

Los resultados obtenidos son los buscados en el objetivo, pues se generan baterías de pruebas con gran facilidad para distintos requisitos de pruebas, y el resultado es fácilmente importado a Excel donde se le puede acomodar el formato a un standard según se necesite, o utilizarlo directamente.

A continuación un ejemplo gráfico de cómo se vé un archivo de salida ya importado a excel:

Gener	Flujo recorrido :	Pasos del Te	v: Selecciona	v: Pago<float>	v: Documento	v: CodigoAfilia	Resultado Esperado :
-ok-	<1>, <1.A>, <1.A.1>, <1.A.1	1) ingresa un	true	1,00	true	D	actualiza la cotizacion
-ok-	<1>, <1.A>, <1.A.1>, <1.A.1	1) ingresa un	true	1,00	true	D	muestra el saldo
-ok-	<1>, <1.A>, <1.A.1>, <1.A.1	1) ingresa un	true	1,68	true	A	actualiza la cotizacion
-ok-	<1>, <1.A>, <1.A.1>, <1.A.1	1) ingresa un	true	1,68	true	A	muestra el saldo
-ok-	<1>, <1.A>, <1.A.1>, <1.A.1	1) ingresa un	true	223,46	true	C	actualiza la cotizacion
-ok-	<1>, <1.A>, <1.A.1>, <1.A.1	1) ingresa un	true	1,11	true	C	muestra el saldo
-ok-	<1>, <1.A>, <1.A.1>, <1.A.1	1) ingresa un	true	54,00	true	C	actualiza la cotizacion
-ok-	<1>, <1.A>, <1.A.1>, <1.A.1	1) ingresa un	true	4,40	true	C	muestra el saldo
-ok-	<1>, <1.A>, <1.A.1>, <1.A.1	1) ingresa un	true	32,45	true	D	actualiza la cotizacion
-ok-	<1>, <1.A>, <1.A.1>, <1.A.1	1) ingresa un	true	34,00	true	D	muestra el saldo
-ok-	<1>, <1.A>, <1.A.1>, <1.A.1	1) ingresa un	-	45,56	true	D	actualiza la cotizacion
-ok-	<1>, <1.A>, <1.A.1>, <1.A.1	1) ingresa un	-	45,56	true	D	muestra el saldo
-ok-	<1>, <2>, <3>, <4>, <5>.	1) ingresa un	-	2,00	true	D	actualiza la cotizacion
-ok-	<1>, <2>, <3>, <4>, <5>.	1) ingresa un	-	2,00	true	D	muestra el saldo

Figura 12. Ejemplo de archivo de pruebas generado por el aplicativo.

DTD para la importación de XMLs

A continuación, se detalla el DTD que debe cumplir cualquier caso de uso en un XML que se desee importar al aplicativo para poder luego generar variaciones de pruebas. Con el solo requisito de cumplir el DTD, se pueden importar casos de uso desde cualquier editor o aplicación que genere casos de uso. El DTD está más simplificado que la gramática de casos de uso del punto 6.1.1 pues muchas cosas de los casos de uso se pueden generar al importarlos, como la numeración de los pasos, debido a la buena elección de los mismos según [Cockburn].

```
<!-- Un caso de uso tiene nombre, version y un flujo basico -->
<!ELEMENT use-case (name, version, basic-flow)>

<!-- el proximo paso de un flujo basico es siempre un secuencial-->
<!ELEMENT basic-flow (sequential-flow-step)>

<!ELEMENT sequential-flow-step
        (
        comment,
        actor,
        action,
```

```
        alternative-flow-step-list,
        next-sequential-or-end-or-goto-flow-step
        )
>

<!-- A los pasos secuenciales solo pueden seguirles mas pasos secuenciales,
pasos de fin o de salto -->

<!ELEMENT next-sequential-or-end-or-goto-flow-step
        (
        sequential-flow-step |
        end-flow-step |
        goto-flow-step
        )
>

<!ELEMENT alternative-flow-step
        (
        comment,
        isErrorAlternativeFlow,
        sequential-flow-step
        )
>

<!ELEMENT alternative-flow-step-list (alternative-flow-step*)>

<!-- El paso de fin de flujo es el unico con post-condiciones y sin paso siguiente -->
<!ELEMENT end-flow-step (post-condition+ )>

<!-- En el paso destino del goto se guarda el step-id con el formato dado x la gram
step-id= numero | numero.letra | numero.letra.step-id -->
```

```
<!ELEMENT goto-flow-step (#PCDATA)>
<!-- Un actor puede ser un usuario o el sistema -->
<!ELEMENT actor (user-actor | system-actor)>
<!ELEMENT user-actor (#PCDATA)>
<!ELEMENT system-actor EMPTY>

<!-- Las acciones son simples o compuestas -->
<!ELEMENT action (simple-action | and-action | or-action)>

<!-- Solo las acciones simples poseen variables de prueba -->
<!ELEMENT simple-action
        (
        description,
        comparation,
        operand,
        variable
        )
>

<!-- Las acciones compuestas AND tienen una lista con dos o mas acciones -->
<!ELEMENT and-action
        (
        description,
        actions-list
        )
>

<!-- Las acciones compuestas OR tienen una lista con dos o mas acciones -->
<!ELEMENT or-action
        (
        description,
```

```
        actions-list
        )
>

<!-- Lista con dos o mas acciones -->
<!ELEMENT actions-list (action, action+ )>

<!-- Todas las comparaciones posibles para valores o rangos de valores -->
<!ELEMENT comparation
        (
        in-comparation |
        less-comparation |
        great-comparation |
        equal-comparation |
        not-in-comparation |
        distinct-comparation |
        less-or-equal-comparation |
        great-or-equal-comparation
        )
>
<!-- Un rango de dominio es en general un subconjunto de un dominio mayor y se
usa como un dominio mas -->
<!ELEMENT range-domain
        (
        character-range-domain |
        date-range-domain |
        float-range-domain |
        integer-range-domain |
        user-defined-domain
        )
>
```

```
<!ELEMENT basic-operand (#PCDATA)>

<!-- Un operando, que es parte de cualquier accion, es simplemente un valor  o es un rango de dominio-->
<!ELEMENT operand (basic-operand | range-domain )>

<!ELEMENT boolen-domain EMPTY>

<!ELEMENT float-domain EMPTY>

<!ELEMENT integer-domain EMPTY>

<!ELEMENT date-range-domain (name, beginning, ending)>

<!ELEMENT float-range-domain (name, beginning, ending)>

<!ELEMENT integer-range-domain (name, beginning, ending)>

<!ELEMENT character-range-domain (name, beginning, ending)>

<!-- Los dominios enumerativos poseen una lista con los valores que los conforman -->
<!ELEMENT user-defined-domain (name, value+ )>

<!ELEMENT domain
        (
        float-domain |
        boolen-domain |
        integer-domain |
        date-range-domain |
        float-range-domain |
```

```
        user-defined-domain |
        integer-range-domain |
        character-range-domain
        )
>

 <!ELEMENT variable (description, domain)>

<!--    TIPOS BASICOS   -->
<!ELEMENT in-comparation EMPTY>
<!ELEMENT less-comparation EMPTY>
<!ELEMENT great-comparation EMPTY>
<!ELEMENT equal-comparation EMPTY>
<!ELEMENT not-in-comparation EMPTY>
<!ELEMENT distinct-comparation EMPTY>
<!ELEMENT less-or-equal-comparation EMPTY>
<!ELEMENT great-or-equal-comparation EMPTY>


<!-- string fields -->
<!ELEMENT name (#PCDATA)>
<!ELEMENT value (#PCDATA)>
<!ELEMENT version (#PCDATA)>
<!ELEMENT comment (#PCDATA)>

<!ELEMENT description (#PCDATA)>
<!ELEMENT post-condition (#PCDATA)>


<!ELEMENT beginning (#PCDATA)>
<!ELEMENT ending (#PCDATA)>
```

7. Conclusiones y Trabajos futuros

7.1. Conclusiones

La aplicación de QA en la Ingeniería de Software aporta un enorme e indiscutido valor, siendo los casos de prueba un pilar fundamental para lograrlo.

Automatizando la generación de pruebas se consigue optimizar el testing logrando mejores resultados en cuanto a tiempos y calidad. Una herramienta para dicha generación de casos de prueba debería estar disponible para cada tester en cualquier equipo de testing. La misma debe ser sencilla de utilizar, fácil de configurar y poder obtener resultados rápidos y en un formato consistente con las demás herramientas que dispone un tester en general.

En este trabajo se estudiaron analizaron diversos tipos y estilos de casos de uso con el resultado de un modelo que contempla los aspectos escenciales de los mismos, para poder manipularlos en la generación de los casos de prueba, los cuales se generan conservando los aspectos también escenciales para cubrir todas las condiciones que se pueden modelar con casos de uso.

Para la generación de casos de prueba se mantiene la premisa de la confiabilidad que se debe obtener siempre en los casos de prueba.

El desarrollo de la herramienta se realizó como un prototipo incremental, corrigiéndose las desviaciones para obtener un prototipo final con fundamento en las decisiones tomadas y que cumpla con las necesidades básicas y comunes, brindando un aporte que intenta contribuir a que la actividad de testing sea más redituable y amigable, y consecuentemente más utilizable.

7.2. Trabajos futuros

Como el alcance de este trabajo se centra en las llamadas pruebas de unidad que son las más habituales e importantes en el testing, no se manejan las extensiones ni las inclusiones de UML, dejándose los mismos notados como acciones textuales. Por ejemplo: ‘includes: ir al CU xxx’.
Una posible extensión a este trabajo sería incluir la implementación de las relaciones de dependencia entre UC (includes y extends) para generar pruebas de integración.

Otra posible extensión sería la implementación de módulos de importación de los casos de uso de herramientas comerciales conocidas para poder luego generar los casos de prueba.

8. Referencias bibliográficas

[Cockburn] Alistair Cockburn, “Writing Effective Use Cases". Addison – Wesley. 21/2/2000

[UML] Modelado de Sistemas con UML
Popkin Software and Systems
http://es.tldp.org/Tutoriales/doc-modelado-sistemas-UML/multiple-html/

[Combinatorial] The Combinatorial Design Approach to Automatic Test Generation, As published in IEEE Software September 1996, pp. 83-87
By David M. Cohen, Siddhartha R. Dalal, Jesse Parelius, and Gardner C. Patton
http://www.argreenhouse.com/papers/gcp/AETGissre96.shtml

[DesignByContract] Test Wizard:Automatic test case generation based on Design by Contract™
Karine Arnout[1] Xavier Rousselot[3] Bertrand Meyer[1], [2]
[1]Chair of Software Engineering, Swiss Federal Institute of Technology (ETH)
CH-8092 Zurich, Switzerland
[2]Eiffel Software
356 Storke Road, Goleta, CA 93117, USA
[3]Amadeus
Sophia-Antipolis, France
http://se.inf.ethz.ch/people/arnout/arnout_rousselot_meyer_test_wizard.pdf

[Dijkstra] O.J. Dahl, E. Dijkstra, and C.A.R. Hoare: *Structured programming*. New York: Academic Press, 1972.

[self-checking] Automatic generation of random self-checking test cases by D. L. Bird C. U. Munoz
http://www.research.ibm.com/journal/sj/223/ibmsj2203G.pdf

[UML-based] UML-based Test Generation and Execution
http://www.gm.fh-koeln.de/~winter/tav/html/tav21/TAV21P6Vieira.pdf

[Rosetta] Automatic Test Case Generator in XML from Specifications in Rosetta
http://www.ittc.ku.edu/research/thesis/documents/murthy__kakarlamudi.pdf

[UML-automatic] Using UML for Automatic Test Generation
http://web.comlab.ox.ac.uk/oucl/research/areas/softeng/TACAS2002.pdf

[UC-test] Extended Use Case Test Design Pattern, By Robert Binder
http://www.rbsc.com/docs/TestPatternXUC.pdf

[op-profile] Generation of test cases from an operational profile of the software. Luc Chiasson and Marc Frappier 8/2/2001
http://www.dmi.usherb.ca/~frappier/gen-test.pdf

[SUMR] SUMR use case editor.
http://www.clearviewtraining.com/CVTPlone/SUMR/SUMRUseCaseEditor

[ePlatero] The ePLATERO Project: “Formal Tool for the Evolutionary Software Development Process”.
Pons, C., Giandini, R, Pérez., G., Pesce, P., Becker, V., Longinotti,J., Cengia,J., Kutsche, R-D.
Home page: http://sol.info.unlp.edu.ar/eclipse., 2003-2005
http://www.oneclipse.com/plugins/education/eplatero/view

[emf] Eclipse Modeling Framework Project
http://www.eclipse.org/emf/

[JavaCC] The JavaCC Tutorial
http://www.engr.mun.ca/~theo/JavaCC-Tutorial

[JavaCC-2] Build your own languages with JavaCC
http://www.javaworld.com/javaworld/jw-12-2000/jw-1229-cooltools_p.html

[DTD] XML DTD - An Introduction to XML Document Type Definitions
http://www.xmlfiles.com/dtd/

[EBNF] Extended Backus Naur Form
Standard EBNF ISO 14977

[XStream] Is a simple library to serialize objects to XML and back again.
http://xstream.codehaus.org/

[Test] IEEE Std 829-1998 IEEE Standard for Software Test Documentation - Description
http://standards.ieee.org/reading/ieee/std_public/description/se/829-1998_desc.html

[RUP] IBM Rational Unified Process
http://www-304.ibm.com/jct03004c/businesscenter/smb/us/en/solutionsummary/xmlid/29811/nav_id/product

9. Apéndice: Definiciones y Acrónimos

UML: Lenguaje Unificado de Modelado. Prescribe un conjunto de notaciones y diagramas estándar para modelar sistemas orientados a objetos, y describe la semántica esencial de lo que estos diagramas y símbolos significan. UML se puede usar para modelar distintos tipos de sistemas: sistemas de software, sistemas de hardware, y organizaciones del mundo real. UML es una consolidación de muchas de las notaciones y conceptos más usados orientados a objetos. Empezó como una consolidación del trabajo de Grade Booch, James Rumbaugh, e Ivar Jacobson, creadores de tres de las metodologías orientadas a objetos más populares.

Caso de Uso (CU o UC): En UML, se tienen a los Casos de Uso para modelar los procesos de negocio. Un caso de uso expresa un contrato entre los stakeholders acerca del comportamiento de un sistema. Describe el comportamiento del sistema y sus interacciones bajo varias condiciones y como responde a los pedidos del actor principal, mostrando como se cumple o falla la meta, y siempre cuidando los intereses de los stakeholders. Un caso de uso junta a todos esos escenarios relacionados con la meta del actor principal.

Actor: cualquiera o cualquier cosa con comportamiento.
Puede ser un sistema mecánico, un sistema de computadoras, una persona, una organización o alguna combinación.
Los actores representan a los usuarios y otros sistemas que interaccionan con el sistema, pero no a las instancias en particular, sino a los tipos de usuario o roles.

Stakeholder: es un actor externo al sistema, que tiene intereses que el sistema debe cuidar, y para los mismos el sistema debe actuar.
Ejemplos de stakeholders son: el dueño del sistema, los directores de la compañía, entes reguladores, como el departamento de seguros, etc.

Actor principal: es un stakeholder que pide al sistema que realice un objetivo. Típicamente pero no siempre es el que inicia la interacción con el sistema.

Escenario: es una secuencia de acciones e interacciones que ocurren bajo ciertas condiciones, expresadas sin condicionales.

Flujo o Camino Básico: es el escenario escrito desde el evento disparador hasta el cumplimiento de la meta. Es típicamente un escenario de éxito más representativo del caso de uso, aunque podría ser de falla.

Flujo alternativo: es cualquier otro escenario o fragmento que es una extensión del flujo básico.

Paso de acción: es la unidad de escritura en un escenario. Típicamente una oración, que usualmente describe el comportamiento de un solo actor.

Paso Secuencial: paso de acción que se sucede en forma secuencial en alguno de los flujos, básico o alternativos.

Paso alternativo: paso de acción que encabeza un flujo alternativo. Un paso alternativo existe si existe un paso secuencial del que se desprende un escenario alternativo.

Paso de fin: último paso de cualquier flujo que no sea un paso de salto. Posee postcondiciones.

Paso de Salto (GOTO): bifurcación a otro paso de alguno de los flujos. Sirve para reutilizar flujos.

Post-condición:
Una post condición es una restricción especifica que se usa para documentar el cambio en condiciones que deben ser verdaderas después de la ejecución del caso de uso.

Acción:
Es el predicado de las oraciones que forman los flujos de los casos de uso.
Puede ser simple o compuesta.

Acción Simple:
Posee una única variable de prueba, y una comparación de la misma contra algún operando, siendo en un caso muy básico igual a verdadero.
Ej.: …. ingresa un importe válido. (importe > 0)

Acción Compuesta AND:
Conjunto de acciones, pudiendo tener cada acción la misma o distintas variables de prueba. Todas deben ser evaluadas con valores que hagan la condición de cada subacción verdadera.

Acción Compuesta OR:
Conjunto de acciones, pudiendo tener cada acción la misma o distintas variables de prueba. Basta que una subacción sea evaluada con algún valor que haga su condición verdadera.

Variables: son los factores que varían de un escenario a otro y determinan las diferentes respuestas del sistema. En general son: las entradas y salidas explícitas, condiciones de entorno que hacen que el actor se comporte diferente, y abstracciones de estados del sistema a probar. Las variables se usan para construir los casos de prueba. Se comienzan con las variables de la interfaz y luego se agregan las demás variables que condicionen los escenarios. [UC-test]

Dominio de variable:
Toda variable tiene un dominio, siendo este el conjunto de valores que puede tomar. Un dominio puede ser entero, booleano, definido por el usuario, etc.

Variación de dominio: valor del dominio de una variable para probar una condición. Por ejemplo: para probar una variable entera con la condición que sea >=10, una variación válida sería el valor 10 mínimamente, luego el 11, y luego algún valor superior.

QA (Quality Assurance / Aseguramiento de la Calidad):
El aseguramiento de la calidad es la actividad de proveer la evidencia necesaria para establecer confianza entre todos los afectados, que las actividades relacionadas a la calidad están siendo realizadas efectivamente.
Todas las acciones deben ser planeadas para proveer la confianza adecuada de que el producto o servicio satisfacerá los requerimientos dados con calidad.
QA es una parte y par del gerenciamiento de la calidad, creando confianza a los clientes externos y a los stakeholders, que el producto coincide con las necesidades, expectativas y otros requerimientos. QA asegura la existencia y efectividad de procedimientos que aseguren los niveles de calidad deseados.
QA debe estar presente en todas las actividades, desde el diseño, el desarrollo, la producción, la instalación y el mantenimiento.

SQA (QA para el software) consiste en procesos de IS y métodos para asegurar calidad. Entre los procesos se incluyen revisiones de documentos de requerimientos, control del código fuente, revisiones de código, manejo de los cambios, manejo de las configuraciones, manejo de las entregas, y por supuesto testing de software.
Notar que testing no es sinónimo de QA, pero sé es una parte fundamental.

Caso de Prueba / Test Case (CP / CT):
Un Caso de Prueba es un documento con las pruebas para testear un software.

Testing humano:
Es el testing ejecutado por personas y no automatizado. Es necesario ya que no todo se puede automatizar, y automatizar tiene a veces más costo e inversión que testear con personas.

Resultado Esperado:
Parte indispensable de una prueba, pues es un texto que indica que es lo que se debe poder comprobar luego de ejecutar la prueba.

Resultado obtenido:
Es el resultado de ejecutar una prueba. Puede ser comprobar un alta en una tabla en base de datos, observar un cartel de error, etc.

Bug / Error:

Una definición de error es la diferencia que se observa entre el resultado esperado y el obtenido. Según esta diferencia, se pueden clasificar desde triviales, menores mayores, críticos, bloqueantes y catastróficos.

Verificación y Validación:

Para el testing de software se utiliza el concepto de V&V (verificación y validación). Verificación es el chequeo o testeo de ítems, incluyendo el software, para la conformidad y consistencia con la especificación asociada. El testing de software es una de las clases de verificación, la cual también usa técnicas como revisiones e inspecciones. Validación es el proceso de chequear que lo que está especificado es lo que el usuario realmente quería. Se llama también el testing de documentación.

XStream: es una librería de Java para serializar y deserializar objetos.
Es fácil de usar y no necesita archivos de mapeos.
Tiene muy buena performance en velocidad y poco uso de memoria.
Brinda un xml limpio siendo mas fácil de leer para humanos, y mas compacto para la serialización de Java, ya que no genera información duplicada.
No necesita modificar a los objetos (no es intrusivo) y serializa campos públicos y privados y clases internas, y sin necesidad de constructores default.

Serialización: proceso de codificación de un Objeto (programación orientada a objetos) en un medio de almacenamiento (como puede ser un archivo, o un buffer de memoria) con el fin de transmitirlo a través de una conexión en red como una serie de bytes o en un formato humanamente más legible como XML. La serie de bytes o el formato pueden ser usados para re-crear un objeto que es idéntico en su estado interno al objeto original (en realidad un clon). La serialización es un

mecanismo ampliamente usado para transportar objetos a través de una red, para hacer persistente un objeto en un archivo o base de datos, o para distribuir objetos idénticos a varias aplicaciones o localizaciones.

EBNF (Extended Backus Naur Form):

En ISO 14977 se define un standard para EBNF, que es una extensión a BNF para ser más clara para el humano y potente en expresividad que BNF, pero es siempre equivalente a la misma, salvo que BNF tiene más producciones.
BNF es una meta sintaxis usada para expresar gramáticas libres de contexto, que es una manera formal de definir lenguajes.